Algoritmi de sortare

-Comparațiile dintre aceștia-

Scopul proiectului

- Sortarea mai multor algoritmi de sortare și compararea timpilor de execuție.
- Au fost aleși 5 algoritmi de sortare
- Au fost ordonate un numar variabil de elemente:
  - 10^3
  - 10^4
  - 10^6
  - 10^7
- Au fost date 3 tipuri de output-uri:
  - Random
  - Reverse
  - Sorted
- Fiecare algoritm a fost rulat de 8 ori și a fost calculat un timp CPU mediu pentru fiecare

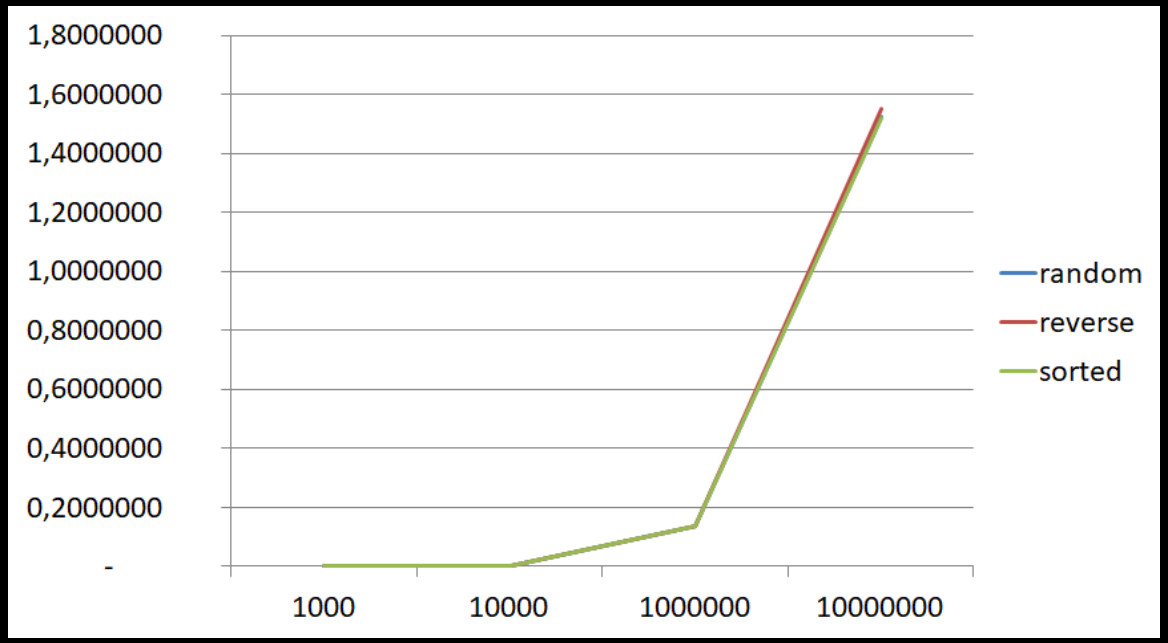
Algoritmii aleși

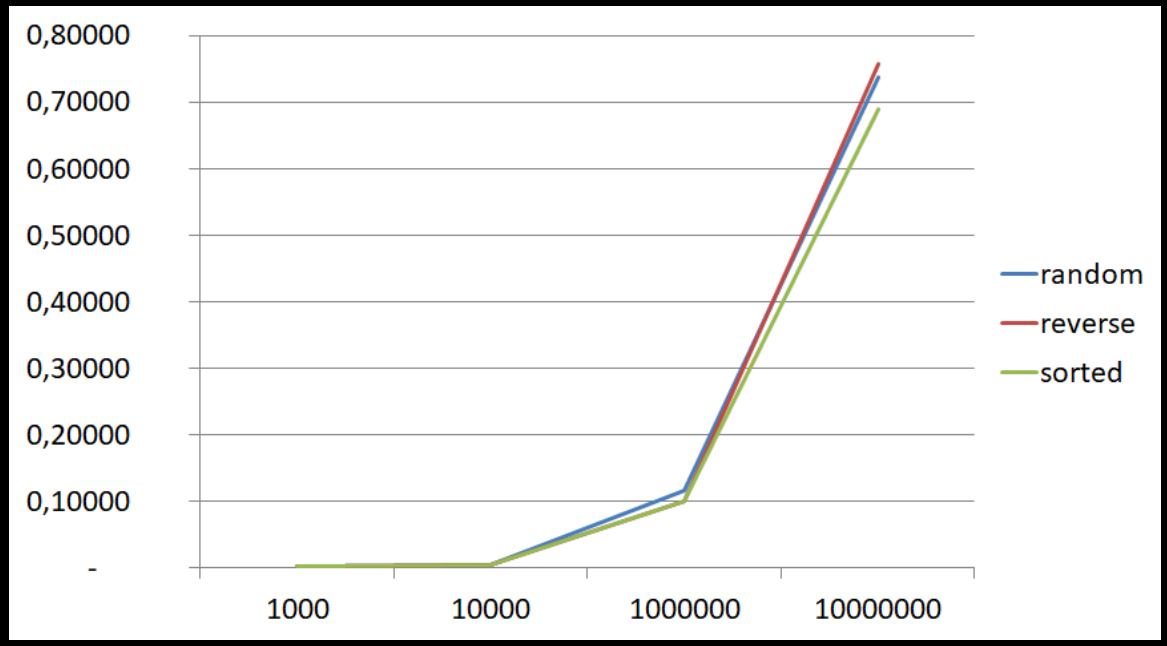
- Radix sort
- Merge sort
- Heap sort
- Shell sort
- Insertion sort

Radix Sort

- Caz defavorabil: O(N log max)
- Memoria folosită: O(N+B)
- Este eficient în sortarea unui număr mare de elemente
- Nu necesită comparații între elementele din lista pentru sortat, de aceea este relativ mai rapid decât alți algoritmi de sortare în cazul unor tipuri specifice de date de intrare.
- Este ineficient în sortarea unui număr mic de elemente. Pentru un număr mic de elemente Merge sort sau Quick Sort sunt mai eficiente
- Ocupă destul de multă memorie

Timpi pentru Radix Sort (baza 2^16)



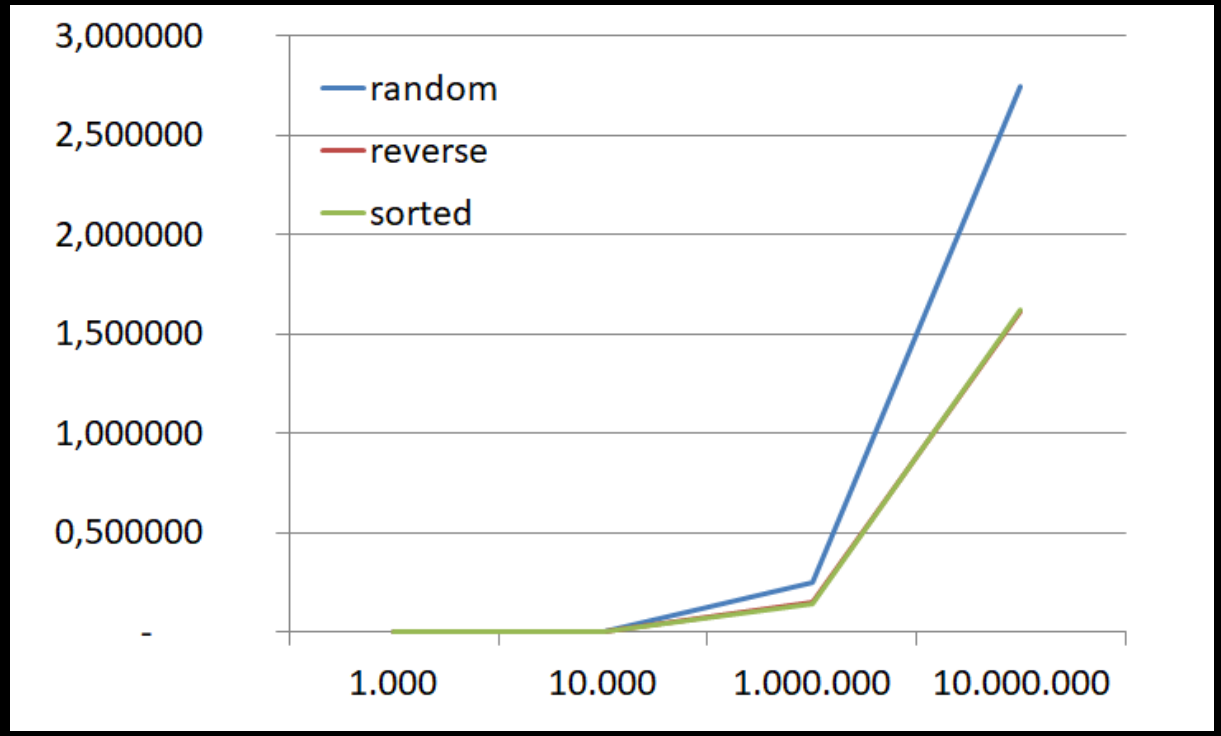


Merge Sort

- Cazul defavorabil: O(N log N)
- Cazul mediu: O(N log N)
- Memoria folosita: O(N)
- Scurtă descriere a algoritmului:
  - Împarte vectorul in secvente din ce în ce mai mici și sorteaza fiecare secvență. Apoi concatenează secvențele înapoi pentru a realiza forma finală.
- Avantajul acestui cod este că are o complexitate de O(N log N) ceea ce înseamnă că poate să sorteze vectori mari relativ rapid. În plus, este o alegere bună pentru sistemele cu resurse limitate, deoarece foloseste puține resurse suplimentare (memoria) pentru sortare.
- Nu este optim pentru date aproape sortate: În cazul în care datele sunt deja aproape sortate, Merge sort poate fi mai lent decât alte algoritmi de sortare, deoarece trebuie să execute pași suplimentari pentru a combina datele. Merge sort necesită spațiu suplimentar de memorie pentru a stoca tablourile intermediare în timpul procesului de sortare.

Timpii pentru Merge Sort

- Observatii:
  - Odata cu cresterea valorilor si timpul creste constant.
  - Timpii de la reverse si sorted sunt apropiati

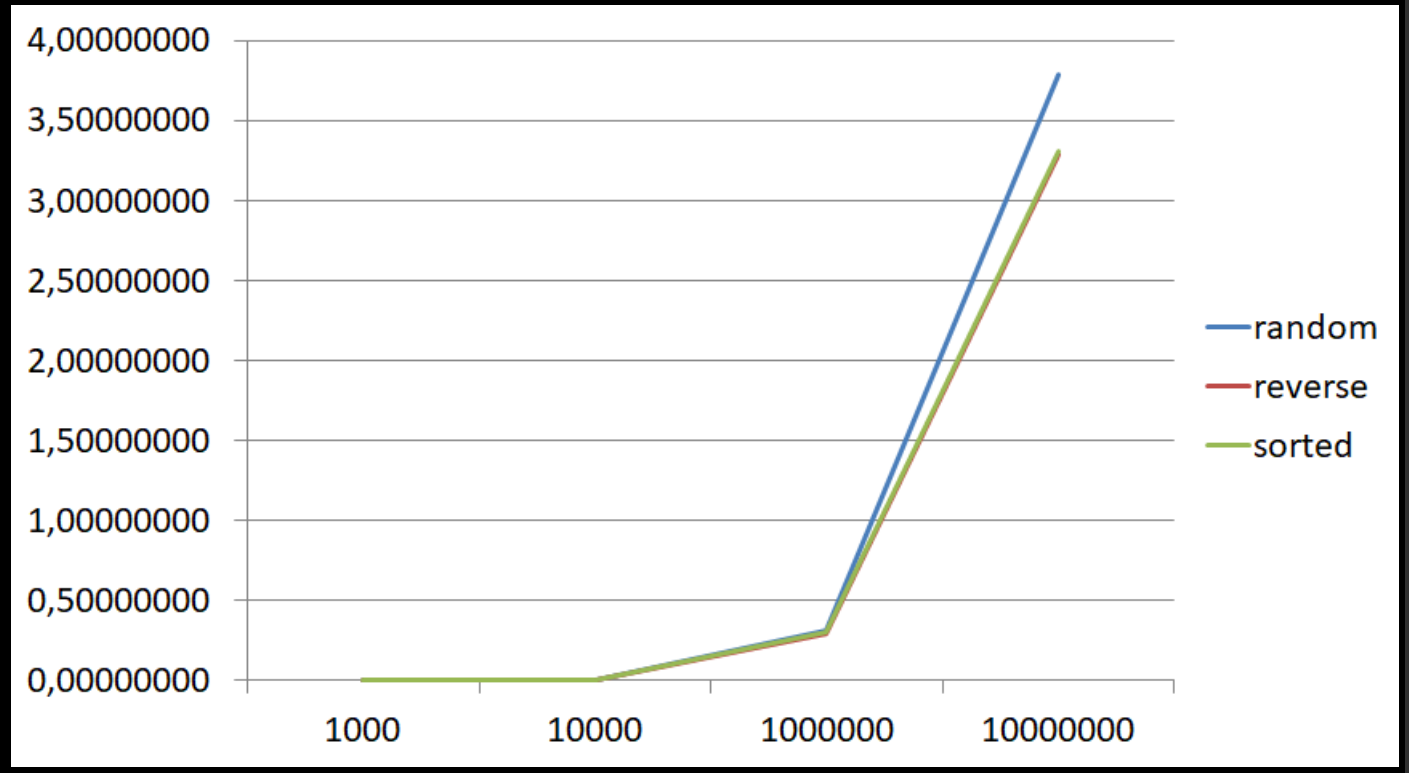


Heap Sort

- Caz favorabil: O(N log N)
- Caz defavorabil: O(N log N)
- Memorie: O(1) -> consum relativ mic de memorie
- Heap-ul = structură de date care permite stergerea, inserarea și acces rapid la maxim sau minim dintr-un set de date
- Nu este optim pentru puține numere
- Este instabil, ordinea relativă a elementelor identice poate să fie schimbată în timpul sortării
- Necesită crearea unei structuri de date suplimentare pentru a stoca datele.

Timpii pentru Heap Sort

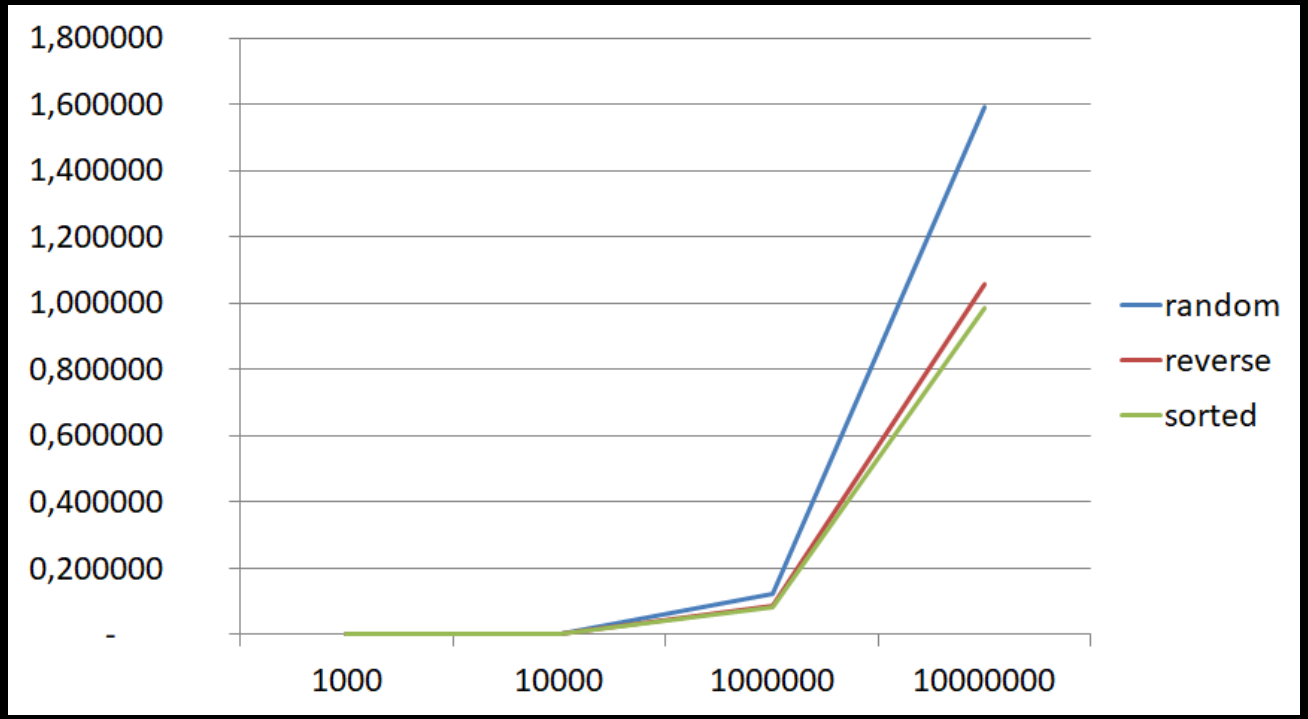
- Se poate observa că timpii dintre reverse si sorted sunt foarte similari
- Timpul crește din ce în ce mai mult (cauzat de crearea structurii de date suplimentare)



Shell Sort

- Caz defavorabil: O(N^3/2)
- Caz favorabil: O(N log N)
- Memorie folosita: O(1)
- Este o generalizare a algoritmului insertion sort. Acesta deplasează elementele spre poziția finală cu mai multe poziții. Se face iterație in care se aplică insertion sort cu gap1 mai mare de 1. Atâta timp cât întâlnește elemente mai mari decât el, elementul din șirul inițial este deplasat spre stânga cu câte gap1 poziții.
- Se repetă iteratiile pentru diferite gap-uri din ce în ce mai mici. Ultima iterație se face cu saltul 1, iar aici este pactic un insertion sort.

Timpii pentru Shell Sort

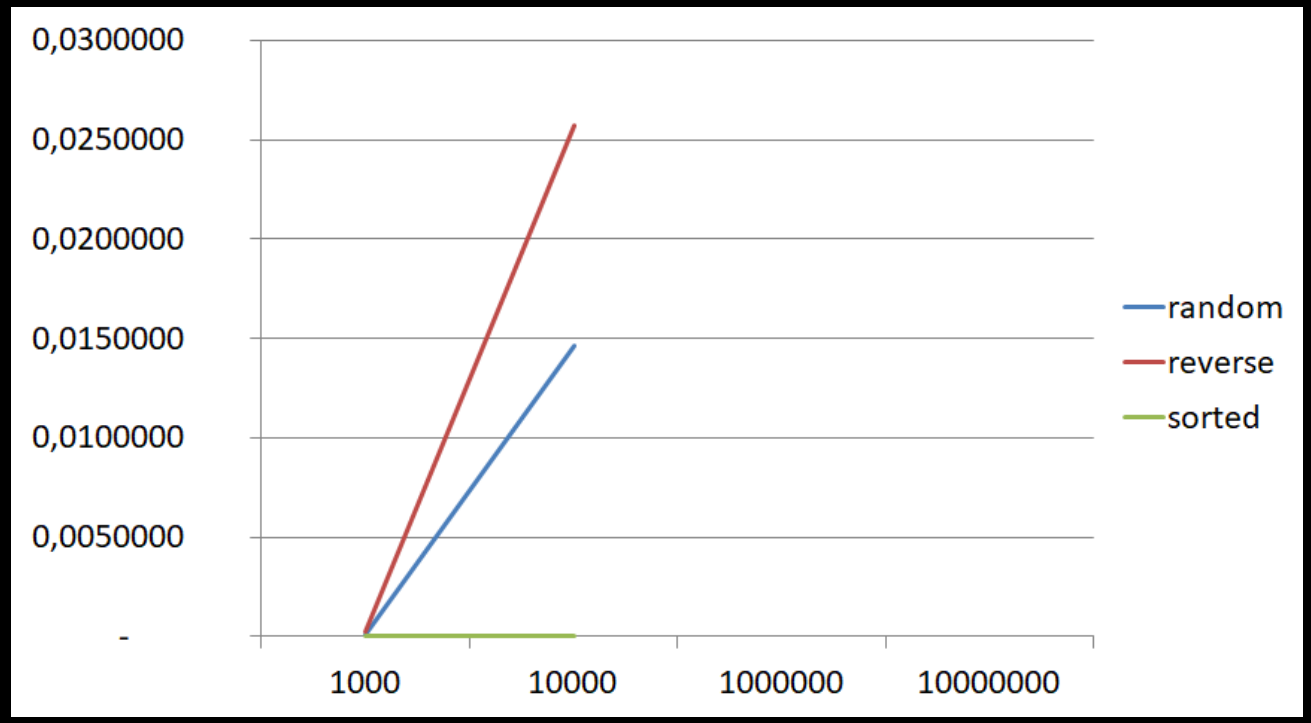


Insertion Sort

- Cazul defavorabil: O(N^2)
- Cazul favorabil: O(N)
- Memoria ocupată: O(1)
- Eficient pentru un număr mic de elemente
- Nu poate fi paralelizat pentru a optimiza timpul de sortare.

Timpi pentru Insertion Sort

- Pentru valori mari precum 10^6, durează foarte mult...



Comparațiile de timpi între algoritmi random order

- Observații:
  - Nu există multe diferențe între algoritmi până la 10^6.
  - Radix Sortul este cel mai rapid pentru numere mari.
  - Timpul la Heap Sort este din ce în ce mai mare